

LLVM Backend for SD-8516

February to April 2026

I started by copying the Lanai files and renaming everything to SD8516. When it compiled I moved forward.

Yes, this took three months.

April and May 2026

During this time I worked on the back-end. There are many tutorials on the 'net such as the CPU0 tutorial. I chose LLVM because I heard the documentation was more useful. Thank God, because I was going to need everything I could get. It's impossible to describe this process. I should have kept more notes on the wiki about it. I may have been able to save myself several days worth of grepping to find random stuff in the target's source tree. This period I call the insanity days. 4 and 8 hour days of doing nothing but trying to lower LLVM code into SDA assembly.

This period is binary. It compiles or it does not. There is no incremental improvement here. I learned a lot and actually changed my ISA a bit because of what I learned doing this. But, I do not remember anything I did here. It is a black hole. I've probably blocked it out of my memory. It was that horrifying.

Notes from April/May

- That stupid string that looks like e-p:32:8-i16:8-i32:8-i64:8-n16 is in:
 - `llvm/lib/TargetParser/TargetDataLayout.cpp`

Total cost of this note alone: Probably 6 hours.

- COMMON files I need to find:
 - `llvm/lib/Target/SD8516/SD8516InstrFormats.`
 - `llvm/lib/Target/SD8516/SD8516InstrInfo.`
 - `llvm/lib/Target/SD8516/SD8516ISelLowering.`

Common advice:

- Look very carefully at the error message. 90% of the time you're missing a header, a function declaration, or something's type is wrong. The solution is often easier than you think.
- Use git or you will mess stuff up and be unable to go back.

June 2026

A sudden thrust of energy, cola and Doritos and the back-end was in sight. A lot of this was fueled by

excitement over many small wins. Such as getting something to work. Once I saw it start to emit SDA assembly I was hooked. Many small changes to the assembler were made too, to support LLVM stuff that I didn't want to try and lower out. One example of an extreme change I made was to remove MOV touching the zero flag because I couldn't figure out how to set it in the files. Its supposed to be [SR] (status register). But there are multiple places you're supposed to add it and I started hallucinating that I had added it in certain places then when I went back to check it was gone. Whatever, I'll just remove the flags from MOV! Now i'm done with that part. HAHA. Hehe hoho.

Suddenly I was out of the tunnel and I didn't even realize it. I decided to skip some stuff that would have made the generated code "better". Like jumptables and some function calls (that are instead inlined, which bloats code).

- Remember: jumptables for large switches
- Remember: mul and div calling functions versus inlining

Notes to Self from June

LLVM is split into a large target-independent middle and a small target-specific back end.

Middle:

- IR optimization,
- SelectionDAG legalization,
- register allocation,
- the MC layer

This doesn't make sense right now, and I don't think it ever will. But read it again and continue.

A full path a C program travels:

- C source
- clang (front end) (C → LLVM IR, applies type model & ABI)
- LLVM IR
- SelectionDAG (IR turned into a per-block dataflow graph)
- legalization (ISelLowering) (illegal ops rewritten into legal ones)
- instruction selection (ISelDAGToDAG) (DAG nodes to machine instructions)
- register allocation (virtual regs → physical regs using the reg file)
- frame lowering (prologue/epilogue, frame indices, SP+offset)
- MC layer (AsmPrinter) (prints... asm...)
- SD-8516 assembler (text to bytes)
- SD-8516 (runs code)

1. The back end (llc target).

Everything from SelectionDAG down to text asm. This is the bulk of the work and can be exercised on hand-written IR long before clang exists. It means you can write an .ll test and use ;CHECK to see if it produces the right assembly. Building a series of tests.

2. The front end (clang TargetInfo).

Teaches the C compiler your type sizes and ABI so it emits IR that your back end already handles. Small, but it has to agree with the back end exactly.

Conventions that had to line up on both sides: `@label` for address references, `#` for decimal, `$` for hex, `.equ` / `.bytes` / `.asciz` / `.zero` / `.long` data directives, and `;` as the comment character.

2. The back end, layer by layer

Files live in `llvm/lib/Target/SD8516/`. The `.td` files are TableGen: a declarative DSL that is compiled at build time into C++ tables (register info, instruction info, the bulk of the instruction-selection matcher). TableGen resolves names strictly top to bottom, so a def must appear before anything that references it.

2.1 Target registration & the TargetMachine

`SD8516TargetMachine.cpp,h` is the top-level object for the target. It wires together the subtarget, the pass pipeline, and the relocation model, and it is what `RegisterTarget` exposes so that `llc -mtriple=sd8516` and `clang` can find the back end. Use the static model to suppress `label$local`.

2.2 The datalayout (and where it actually lives)

The `datalayout` string describes endianness, pointer width/alignment, integer alignments, and native integer widths; the `e-p:32:8-i16:8-i32:8-i64:8-n16` crap from before.

e-p:32:8-i16:8-i32:8-i64:8-n16 means little-endian; 32-bit pointers, byte-aligned; i16/i32/i64 all byte-aligned; native width 16. It drives struct layout and `alloca` alignment.

If you omit a type from the string, LLVM uses a built-in default that may not be correct.

- Before, i64 was absent, so it defaulted to 4-byte alignment (an x86-32 legacy), which later collided with the front end.

2.3 Registers

in `SD8516RegisterInfo.td`

Defines the physical registers in register classes. On SD8516 the 32-bit pairs are built from 16-bit halves (`AB = A:B`, and so on), which is the source of the recurring “clobbergoblin” hazard: writing a half silently disturbs the pair, and vice versa.

2.4 Instruction formats and instructions

in `InstrFormats.td` and `InstrInfo.td`

`InstrFormats.td` holds base classes describing the “shape” of instructions. I don't know what that means. Put `let Defs = [SR];` on the flag-writing format classes (ex. LD affects Z and N but MOV was changed not to touch Z or N).

`InstrInfo.td` is the actual instruction list. Each instruction names its operands, its assembly from `AsmPrinter`, and usually a selection pattern (DAG written in `TableGen`) that says “when you see this shape of computation, use me.” Also used for pseudo-instructions `SELECT_CC` or `LEA_FI` that aren't real opcodes but get expanded later by C++.

2.5 Calling conventions

`SD8516CallingConv.td`

A declarative table mapping argument and return values to registers and stack slots, by type. Ours assigns `i16` to single registers (A, B, X, Y), `i32` (pointers) to the pairs (AB, CD, XY, IJ), overflow to the stack, and returns `i16` in A / `i32` in AB.

2.6 Operation legalization & lowering

`SD8516ISelLowering.cpp`

The brain. This is `TargetLowering`, and it's where most of the real decisions live. Its constructor declares, per IR operation and type, one of:

- **Legal** the machine does this directly.
- **Expand** LLVM should rewrite it into smaller legal operations (ex lower `i64` into pairs of `i16`)
- **Custom** call your C++ to rewrite it (`SELECT_CC`, `BR_CC`, `varargs`, and signed divide).
- **LibCall** a call to a runtime helper (simulating a fpu, wide divides/multiplies, routing to `__addsf3`, `__divsi3`, etc).

This is what implements the ABI: `LowerFormalArguments`, `LowerCall`, `LowerReturn` consume `CallingConv.td` tables to move values into the right registers and stack slots, and `LowerOperation` holds the custom rewrites.

2.7 Instruction selection

`SD8516ISelDAGToDAG.cpp`

After legalization leaves a DAG made only of legal operations. Instruction selection walks that DAG and replaces each target-independent node with one of the actual machine instructions. The patterns from `InstrInfo.td` are compiled by `TableGen` into an automatic matcher.

The `.cpp` exists for the parts `TableGen` can't express as a declaration, ex. addressing-mode selection. For that we have a hand-written routine (a `ComplexPattern`, e.g. a `SelectAddr`) to decide how to fold a base register, an index, and a displacement into a memory operands. Its entry point is `Select()`, with optional pre/post-processing hooks. Mental model: `ISelLowering` decides what legal operations exist.

ISelDAGToDAG decides which machine instruction implements each one.

Glue

InstrInfo.cpp, RegisterInfo.cpp, FrameLowering.cpp

These implement hooks the generic code calls during and after selection:

- **SD8516InstrInfo.cpp** (TargetInstrInfo): copyPhysReg (how to emit a register copy ex. MOV), spill/reload helpers, branch analysis
- **SD8516FrameLowering.cpp** (TargetFrameLowering): emitPrologue/emitEpilogue handles SUB SP, #n and ADD SP, #n and any frame-pointer.

2.9 Emitting text

MCAsmInfo the instruction printer

The MC layer turns selected, register-allocated instructions into sda assembly. MCAsmInfo describes the dialect (comment string ;, directive spellings, label rules); the instruction printer formats each instruction using the assembly strings from InstrInfo.td. This is where output has to match the assembler.

—

4.1 Triple arch

Triple identifies the architecture as a string and an ArchType enum entry.

llvm/include/llvm/TargetParser/Triple.h (the enum) and .../Triple.cpp (the name/parse/width functions) must know `sd8516`, or clang rejects the triple outright.

The best way to add it is to mirror an existing (Lanai or MCP430?) small in-tree target across every spot.

4.2 TargetInfo

clang/lib/Basic/Targets/SD8516.h, cpp is a TargetInfo subclass. It declares IntWidth, LongWidth, PointerWidth, and alignments.

SizeType - predefined macros, va_list

That datalayout string must be **byte-identical** to the back end's.

Type Model

int=16 (period-authentic for a 16-bit machine, smallest code, matches the n16 native width, and free

given the calling convention already routed it). For the SD8516, just change the IntWidth.

The build/test loop

- `ninja llc` for back-end changes, `ninja clang` for front-end ones.
- `clang -target=sd8516 -S -emit-llvm test.c -o - | llc -mtriple=sd8516 -o -`

Stuff to Remember

This is LLVM from April 2026. Signatures move.

The datalayout exists in two places and they must match: `TargetDataLayout.cpp` is not in the SD8516 tree.

Use [SR] to mark ops that change status register (i.e. flags)

$A \geq B == C$ (6502-style), inverted from x86 habit.

From:
<https://www.appledog.ca/wiki/> - Appledog

Permanent link:
https://www.appledog.ca/wiki/doku.php?id=sd:llvm_backend_for_sd-8516&rev=1781145974

Last update: **2026/06/11 02:46**

